To the COMAL Development Group.

A PROPOSAL FOR PACKAGES

In the following packages will be denoted by modules.

PURPOSE

The purpose of modules is twofold:

* to allow the specification of logically related procedures, functions, and their data structures.

* to conceal and protect the inner workings of these entities from their users.

SYNTAX

The syntax description below lets the concept of modules enter in the same way as procedures and functions. Modules can only be declared in the outermost (main) scope level.

There may be declared any number of modules in a program.

Within each module is specified which procedures and functions can be called by the user. Any other named entity in a module is private to the module.

If identifiers in the calling procedure, function, module, or program are conflicting with identifiers in a module, then dot-notation may be used to qualify a call.

```
<structured declaration statement> ::=
        <procedure declaration> |
        <function declaration> |
        <module declaration>

<module declaration> ::=
        <internal module declaration> |
        <external module declaration>

<internal module declaration> ::=
        MODULE <module identifier> <eol>
         {<export statement>}
          <module block>
        ENDMODULE <module identifier> <eol>

<external module declaration> ::=
        MODULE <module identifier> EXTERNAL <file name>
```

```
<module block> ::=
        {<non-declaration statement> |
         <procedure declaration> |
         <function declaration> |
         <unstructured declaration statement>}

<export statement> ::=
        EXPORT <export identifier> {,<export identifier>}

<module identifier> ::= <identifier>

<export identifier> ::= <procedure identifier> |
                        <function identifier>

<file name> ::=         <string constant>

<procedure call statement> ::=
        [EXEC] [<module identifier>.]<procedure identifier>
        [<actual parameter list>]

<numeric function call> ::=
        [<module identifier>.]<numeric identifier>
        [<actual parameter list>]

<string function call> ::=
        [<module identifier>.]<string identifier>
        [<actual parameter list>][(<substring specifier>)]

<unstructured declaration statement> ::= <use statement>

<use statement> ::=
        USE <module identifier> {,<module identifier>}
```

SCOPE RULES

First a definition:

By the term "locally declared identifier" is meant an identifier, which in a given scope is declared as a procedure, function, label, array, parameter, or in the case of implicitly declared variables, is assigned either by :=, READ, INPUT, or by call-by-reference.

Within a module:

* A module is closed apart from all other module identifiers, that are not locally declared.

* The module identifier itself is visible, unless it is declared locally.

* A module can refer to its EXPORTed identifiers as well as its private identifiers.

Outside a module:

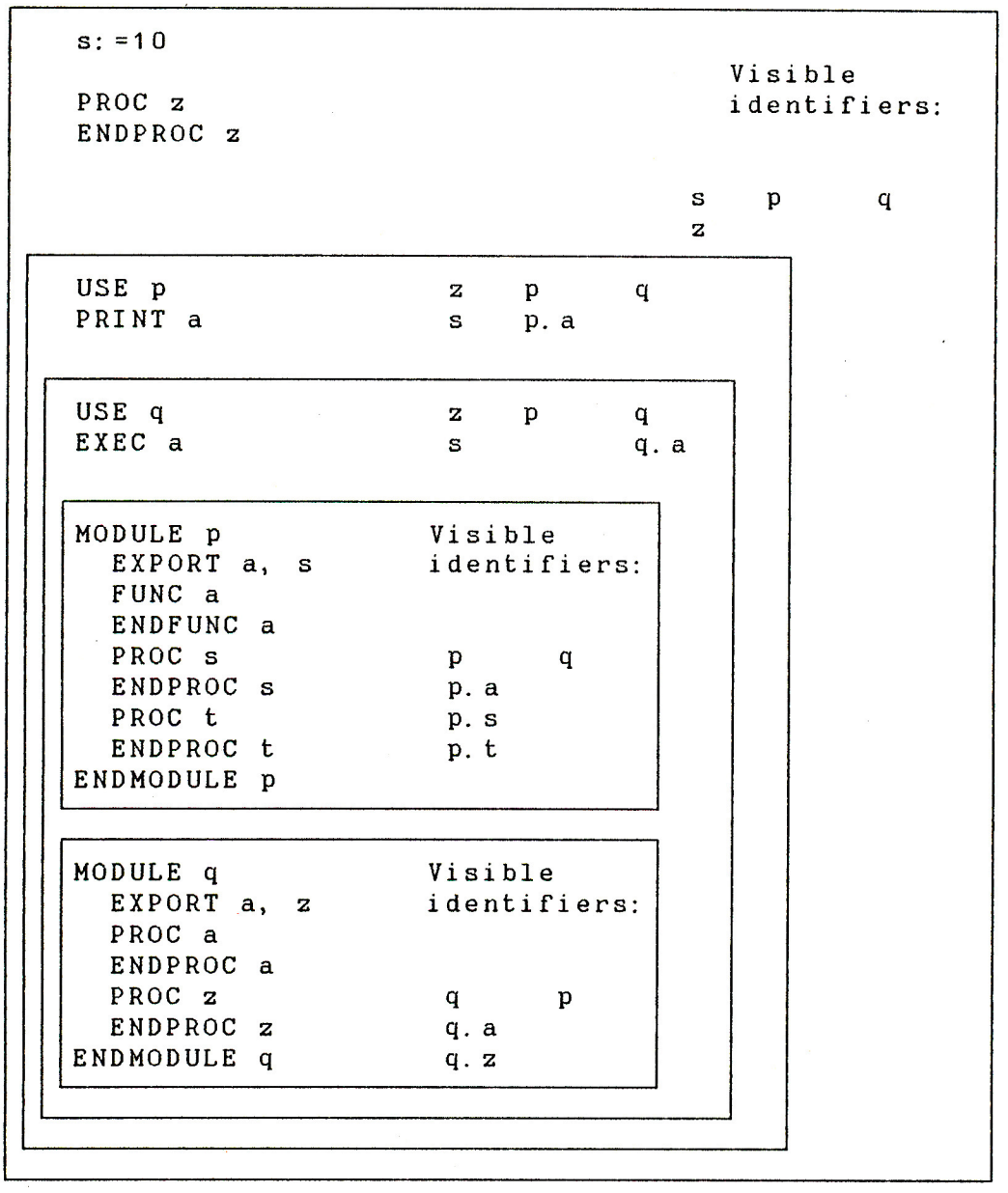
* The module identifier is visible in all scope levels. If the module identifier is declared locally in a scope level, it is not visible in that scope level.

The USE-statement establishes a module's EXPORTed identifiers as visible from the next statement till the last statement in the block. USE-statements may appear in the main program, procedures, functions, and modules.

A USE-statement with more than one identifier corresponds to the same number of USE-statements with only one identifier.

Conflicts among identifiers. Locally declared identifiers always take priority over module identifiers and USEd identifiers. Conflicts among EXPORTed identifiers may be resolved by using qualification.

Below is an example showing the scope rules. When the notation P.A is used, it means that an unqualified A is interpreted as P.A .



SEMANTICS

Before the program is run, all the modules, which are USEd in the main program, and the in modules, which are USEd, are initialised. Initialisation is done by executing a module's <module block>.

The variables in the USEd modules exist as long as the program is running.

ENVIRONMENT

Here is described some of the features every COMAL system can have. These features are not part of the language.

An EXTERNAL module consists of a program containing nothing but a module. It can be created by SAVEing such a program. External modules may be written in COMAL or (if a system can support it) in some other language (assembly Language, Pascal, C, etc.).

Before the program is run, all modules referenced, are placed in memory. This can be done in several different ways:

* Some modules are part of the COMAL system. There are no need to specify them as EXTERNAL.

* The COMAL system may have placed some frequently used modules in a special subdirectory. If a module is not declared in the program, it is searched in this subdirectory. Alternatively, the system have an information file, telling which modules are known. Before the program is run, these module files are loaded, and bound to the program. A compiling system may have a standard library of modules.

* A module is declared EXTERNAL. It is loaded, and bound before the program is run.

* The COMAL system supplies a command for loading modules. These modules may be referenced by MODULE ... EXTERNAL statements or implicitly by USE.

* Loaded modules may be treated as part of the program, i.e. they are stored together with the program when SAVEd and loaded.

The environment will have commands for:

* Discarding modules from memory.

* Showing which modules are placed in memory.

* Showing which procedures, and functions, that are EXPORTed from a given module, as well as their parameters.

* Protecting the source text of a module.

* Editing modules.

In an interactive COMAL-system, it will be useful if the USE-statement can be used as a command. If the module to USE is not in memory, it will be loaded, automatically.

EXAMPLES

1) A stack module.

```
MODULE stack
  EXPORT push, pop, empty

  // Initialize: //

  max:=100    // stack size
  sp:=1       // stack pointer
  DIM s(max)  // stack area

  // Procedures: //

  PROC push(x)
    IF sp>max THEN
      STOP "Stack Overflow"
    ELSE
      s(sp):=x; sp:=sp+1
    ENDIF
  ENDPROC push

  FUNC pop
    IF sp=1 THEN
      STOP "Stack Underflow"
    ELSE
      sp:=sp-1
      RETURN s(sp)
    ENDIF
  ENDFUNC pop

  FUNC empty
    RETURN sp=1
  ENDFUNC empty

ENDMODULE stack
```

2) A queue module.

```
MODULE queue
  EXPORT push, pop, empty

  // Initialize: //

  max:=100    // queue size
  in_ptr:=1   // next place to push
  out_ptr:=1  // next place to pop
  DIM q(max)  // queue area
```

```
  // Procedures: //

  PROC push(x)
    q(in_ptr):=x; in_ptr:=in_ptr+1
    IF in_ptr>max THEN in_ptr:=1
    IF in_ptr=out_ptr THEN
      STOP "Queue Overflow"
    ENDIF
  ENDPROC push

  FUNC pop
    IF in_ptr=out_ptr THEN
      STOP "Queue Underflow"
    ELSE
      x:=q(out_ptr); out_ptr:=out_ptr+1
      IF out_ptr>max THEN out_ptr:=1
      RETURN x
    ENDIF
  ENDFUNC pop

  FUNC empty
    RETURN in_ptr=out_ptr
  ENDFUNC empty

ENDMODULE queue
```

3) Use of STACK and QUEUE.

```
// Declare EXTERNAL modules: //

MODULE stack EXTERNAL "\mod\stack"
MODULE queue EXTERNAL "\mod\queue"

USE stack                   // Establish visibility to stack
REPEAT                      // Until 0 is entered //
  INPUT x
  IF x<>0 THEN push(x)      // stack.push(x)
UNTIL x=0

USE queue                   // Establish visibility to queue
WHILE NOT stack.empty DO    // Move stack to queue //
  push(stack.pop)           // queue.push(stack.pop)
ENDWHILE
```

4) Modules using modules are useful. In some system there will be modules like:

```
VDI             (Virtual Device Interface, a low level graphic tool)
GKS             (Graphic Kernel System, a higher level graphic tool)
TURTLE          (A turtle graphic module)
```

In this case TURTLE is using GKS, and GKS is using VDI:

```
MODULE vdi
  EXPORT ...
  ..
ENDMODULE vdi

MODULE gks
  EXPORT ...
  USE vdi
  ..
ENDMODULE gsk

MODULE turtle
  EXPORT ...
  USE gks
  ..
ENDMODULE turtle
```

If a user wants to use TURTLE, all he/she needs to do, is to include the statement

```
USE turtle
```

in the program. COMAL will find out, that the modules GKS, and VDI also are needed, and load them automatically before the program is run.

RC,
UniComal,
DDE